DCC045 - Teoria dos Compiladores - Relatório Partes I e II

Grupo: Giovane Nilmer de Oliveira Santos - 201835012
Marcos Mateus Oliveira dos Santos - 201835019

Parte I - Análise Léxica com JFlex

Introdução

Neste relatório, abordamos o processo de implementação de um analisador léxico utilizando a ferramenta JFlex. Serão fornecidas informações sobre compilação e execução do código, e também o papel de cada classe neste processo.

Analisador Léxico

O arquivo principal do analisador léxico é o `LangLex.jflex`, contendo as regras que definem os *tokens* reservados, de comando, identificadores de valores, *tokens* lógicos e de símbolos, além de definir as expressões da linguagem e as regras de comentários dentro do código:

Definição das Expressões da Linguagem

As expressões abaixo são utilizadas para identificar padrões que correspondem a diferentes tipos de *tokens*. Essas expressões são fundamentais para categorizar os elementos léxicos corretamente.

1. **numInteger**: `[:digit:]+`
    - **Descrição**: Define um **número inteiro**.
2. **numFloat**: `([:digit:]* "." ([:digit:] [:digit:]*));`
    - **Descrição**: Define um **número float**.
3. **letter**: `[:letter:]`
    - **Descrição**: Corresponde a qualquer **letra do alfabeto**, maiúscula ou minúscula.
4. **id**: `[:lowercase:] ({letter} | "_" | [:digit:])*`
    - **Descrição**: Define um **identificador** na linguagem.
5. **nameType**: `[:uppercase:] ({letter} | "_" | [:digit:])*`
    - **Descrição**: Semelhante a `id`, mas começa com uma letra maiúscula (`[:uppercase:]`) e também pode ser seguida por letras, underscores ou dígitos. É usada para definir **tipos de dados** ou **nomes de classes**.

6. **endOfLine**: `\r|\n|\r\n`
    - **Descrição**: Define o **final de linha**.
7. **whiteSpace**: `{endOfLine} | [ \t\f]`
    - **Descrição**: Identifica **espaços em branco**, incluindo tanto os terminadores de linha (`{endOfLine}`) como os espaços comuns (`" "`), tabulações (`\t`), e form feed (`\f`).
8. **lineComment**: `"--" (.)* {endOfLine}`
    - **Descrição**: Esta expressão define **comentários de linha**. Assim, todo o conteúdo do comentário é ignorado durante a análise léxica.

Tipos de Tokens

- **ID**: Identificadores
- **INT_VAL**: Valores dos inteiros
- **CHAR_VAL**: Valores dos caracteres
- **FLOAT_VAL**: Valores dos booleanos
- **NAME_VAL**.

Palavras Reservadas (CMD):

- **IF**: `if`
- **ELSE**: `else`
- **ITERATE**: `iterate`
- **READ**: `read`
- **PRINT**: `print`
- **RETURN**: `return`

Operadores e Símbolos (EXP):

- **EQUALITY_SIGN**: ==
- **NOT_EQUAL_SIGN**: !=
- **LESSER_THAN**: <
- **GREATER_THAN**: >
- **AND_SIGN**: &&
- **PLUS_SIGN**: +
- **MINUS_SIGN**: -
- **MULT_SIGN**: *
- **DIVIDE_SIGN**: /
- **MOD_SIGN**: %
- **NOT_SIGN**: !

- **EQUAL**: =
- **AMPERSAND**: &
- **DOT**: .
- **COMMA**: ,
- **SEMICOLON**: ;
- **COLON**: :
- **DOUBLE_COLON**: : :
- **OPEN_PARENT**: (
- **CLOSE_PARENT**:)
- **OPEN_BRECKET**: {
- **CLOSE_BRECKET**: }
- **OPEN_BRACE**: [
- **CLOSE_BRACE**:]
- **SINGLE_QUOTES**: '
- **BACK_SLASH**: \
- **EOF**: `\r` | `\n` | `\r\n` (Fim de linha)

Palavras Reservadas - Tipos (RESERVADO):

- **TRUE**: `true`
- **FALSE**: `false`
- **NULL**: `null`
- **INT**: `int`
- **CHAR**: `char`
- **BOOL**: `bool`
- **FLOAT**: `float`
- **DATA**: `data`
- **NEW**: `new`

Compilação e Execução

Execução

- Build do projeto:

```
make run-lexer
```

- Rodar o projeto:

```
java -cp classes LangCompiler <ARQUIVO_TESTE>
```

Parte II - Análise Sintática e Interpretador

Para a segunda etapa do desenvolvimento do compilador da linguagem ‘lang’, optou-se pelo uso do ANTLR4, que possibilita a geração de analisadores léxicos e sintáticos.

Neste contexto, o ANTLR4 transforma a especificação da gramática (definida no arquivo ‘lang.g4’) em uma série de classes Java que representam a AST, na qual cada nó é uma instância de uma classe que representa uma construção específica da linguagem ‘lang’. Essas classes são instanciadas automaticamente pelo ANTLR4 a partir da gramática definida, seguindo um modelo orientado a objetos e gerando a árvore.

Especificação da Gramática

prog: Define a estrutura de um programa, que vai conter múltiplas declarações “data” e definições “fun”, e é uma instância de Program.

data: Define estrutura do tipo Data.

decl: Descreve uma declaração dentro de um bloco de data, especificando um identificador e seu tipo, terminando com um ponto e vírgula. Corresponde à regra DataDeclaration.

fun: Define uma função com um identificador, parâmetros opcionais (params?), tipos de retorno opcionais, e um bloco de comandos. É tratada como uma instância da regra Function.

Tipos e Expressões

type e **btype**: Especificam tipos básicos e tipos de array. Os tipos básicos incluem INT_TYPE, CHAR_TYPE, BOOL_TYPE, FLOAT_TYPE, e NAME_TYPE, enquanto os tipos de array são representados pelo tipo base, seguido de colchetes (ex. Int[]).

cmd: Inclui várias formas de comandos como blocos de comandos, condicionais (if, else), laços (iterate), operações de leitura (read), impressão (print), retorno de valores (return) e atribuições. Cada comando é tratado como uma instância específica de regra, como CommandList, IfCommand, IterateCommand, etc.

Expressões e Operadores

exp, **rexp**, **aexp**, **mexp**, **sexp**: Estas regras definem a precedência e agrupamento de expressões aritméticas e lógicas, desde operadores lógicos (AND_SIGN) até operadores matemáticos (PLUS_SIGN, MULT_SIGN).

Valores Literais e Identificadores

Lexemas: Regras como INT_VAL, CHAR_VAL, FLOAT_VAL definem padrões para literais numéricos, caracteres e flutuantes.

Identificadores: ID define o formato de identificadores variáveis e NAME_TYPE para tipos definidos pelo usuário.

Símbolos e Delimitadores

Definem os símbolos utilizados na linguagem (AND_SIGN, EQUAL, etc.) e delimitadores como parênteses, colchetes, e chaves.

Comentários e Espaços em Branco

Comentários: Regras para ignorar comentários de linha (LINE_COMMENT) e comentários de múltiplas linhas (MULTILINE_COMMENT).

Espaços em Branco: Espaços e novas linhas são ignorados para não afetar a análise sintática.

Funcionamento do LangVisitor

O LangVisitor percorre a árvore de análise sintática gerada pelo ANTLR, onde cada nó representa uma construção da linguagem, como as expressões, comandos, declarações de variáveis, definições de funções, etc.

Cada método “visit” na classe LangVisitor é responsável por um tipo específico de nó da gramática. Por exemplo, um método pode lidar com nós que representam expressões matemáticas, enquanto outro pode lidar com estruturas de controle como loops e condicionais.

Ao visitar um nó, o método correspondente transforma esse nó da árvore de análise em um nó correspondente na AST. Esta transformação geralmente envolve converter a informação contida no nó (como tokens ou subárvores) em uma estrutura abstrata, mais adequada para operações de compilação ou interpretação.

A classe LangVisitor utiliza o padrão de design Visitor, o que facilita a adição de novas operações à AST sem modificar as classes dos nós. Cada nó da AST implementa uma interface Visitable, que inclui o método accept para aceitar (ou seja, validar) um visitante (Visitor).

A interface Visitor, por sua vez, define um conjunto de métodos “visit”, cada um projetado para lidar com um tipo específico de nó da AST.

Exemplo de Método visit:

```
@Override
public Node visitFunction(LangParser.FunctionContext ctx) {
    // Extrai informações do contexto do parser
    String functionName = ctx.ID().getText();
    List<Parameter> params = extractParameters(ctx.params());
    Block body = (Block) visit(ctx.block());

    // Cria um nó de função na AST
      FunctionNode functionNode = new FunctionNode(functionName,
params, body);
    return functionNode;
}
```

Interpretador

A classe LangVisitorInterpreter extende a implementação da LangVisitor e contém métodos para visitar cada tipo de nó da AST, processando e interpretando os diferentes elementos da linguagem definida, como expressões, comandos e declarações de funções.

Estrutura de Dados do Interpretador

```
   private HashMap<String, Data> datas; // HashMap para armazenar
toda as datas do programa
    private HashMap<String, Function> functions;// HashMap para
armazenar toda as funções
     private Stack<HashMap<String, Object>> env;// Pilha que
representa o escopo, armazena nomes de variáveis e seus valores
atuais.
    private Stack<Object> operands; //Pilha usada para avaliar
expressões e armazenar resultados intermediários
  private Stack<Object> params; // Pilha para gerenciar parâmetros
de chamadas de função.
  private boolean modeDebug, retMode; //modeDebug: flag para gerir
detalhes para debug - retMode: flag para gerenciar saídas de
função.
```

Lógicas Utilizadas

Dentre as lógicas utilizadas, selecionamos as que houve maior dificuldade para a implementação, e as definiremos a seguir:

Iterate: Gerencia dois tipos de loops na linguagem, o Loop Condicional que avalia uma expressão booleana e executa o bloco de comandos enquanto essa condição for verdadeira. Após cada execução do bloco, a condição é reavaliada para decidir se o loop continua; e o Loop Contável, no qual se a condição inicial é um número inteiro, trata esse número como a quantidade de vezes que o bloco de comandos deve ser executado. O bloco é repetido esse número específico de vezes.

AssignCommand:

- Avaliação da Expressão:
    - A expressão à direita da atribuição é avaliada e seu resultado é armazenado na pilha de operandos.

- Tratamento de Atribuição para Diferentes Tipos de LValue:
    - Variáveis Simples (IDLValue): O valor obtido é atribuído à variável correspondente, com verificação de compatibilidade de tipo.
    - Propriedades de Objetos (DotLValue): Se a propriedade é de um objeto comum, o valor é diretamente atribuído. Se for parte de um array, o índice é avaliado para atualizar o valor no array.
    - Acesso a Arrays (ArrayAccess): Direciona a atribuição para uma posição específica de um array, garantindo que o índice esteja dentro dos limites válidos do array.

ArrayAcess:

- Validação do Array:
    - Primeiro, tenta recuperar o array do ambiente de execução usando o identificador fornecido.
    - Se o array não for encontrado, lança uma exceção indicando que o array não foi declarado.
    - Verifica se o objeto recuperado é realmente um array.
- Avaliação do Índice:
    - A expressão que determina o índice do array é avaliada.
    - O resultado, esperado ser um inteiro, é retirado da pilha de operandos. Se não for um inteiro, uma exceção é lançada.
- Verificação de Limites:
    - Verifica se o índice calculado está dentro dos limites válidos do array (não negativo e menor que o tamanho do array).
    - Se o índice estiver fora dos limites, lança uma exceção.
- Acesso ao Elemento:
    - Recupera o elemento do array no índice especificado e empilha esse valor na pilha de operandos para uso subsequente.

Representação da AST

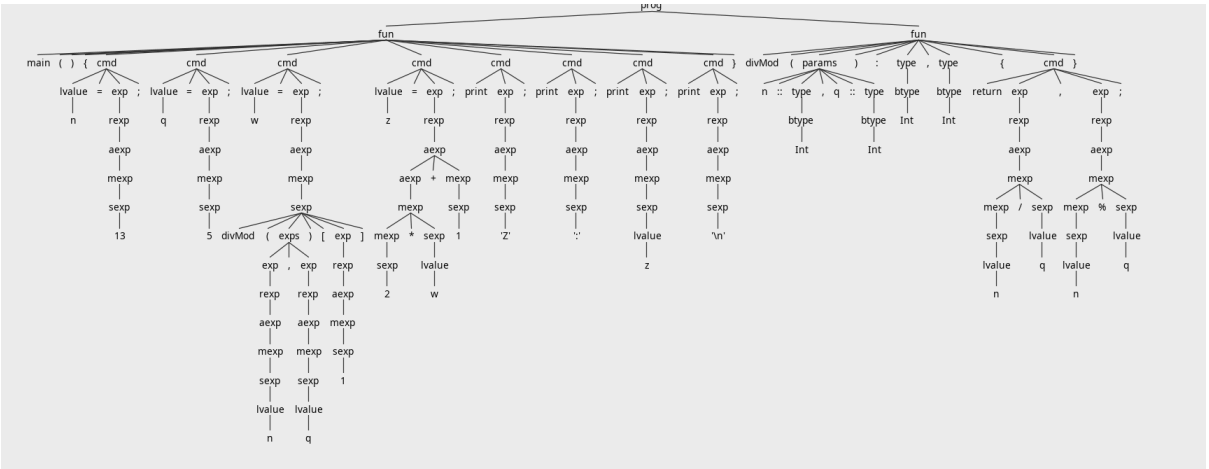
Considerando o seguinte código de entrada:

```
main(){
    n = 13;
    q = 5;
    w = divMod(n,q)[1];
    z = 2*w +1;


    print 'Z';
    print ':';
    print z;
    print '\n';
}

divMod(n :: Int, q :: Int ) : Int, Int{
    return n / q, n % q;
}
```

A AST gerada seria da forma:



Estrutura de Diretórios, Compilação e Execução

Estrutura de Diretórios

1. `lib/`: Contém bibliotecas necessárias, incluindo o ANTLR4.
2. `lang/`: Contém o código-fonte do compilador, incluindo os diretórios `parser`, `interpreter`, e `ast`.
3. `parser/`: Código gerado pelo ANTLR e classes de parser.
4. `interpreter/`: Classes responsáveis pela interpretação dos códigos.
5. `ast/`: Classes que representam os nós da Árvore de Sintaxe Abstrata.

Compilação

O projeto usa um Makefile para simplificar o processo de construção. Os seguintes comandos estão disponíveis:

Compilar Todo o Projeto

```
make all
```

Limpa caches antigos, gera o parser a partir do arquivo `.g4`, compila a AST, o parser, o interpretador e o código principal.

Limpar Cache

```
make clean-cache
```

Limpa arquivos de classes e caches gerados anteriormente.

Gerar o Parser com ANTLR

```
make gen-parser-antlr
```

Gera o código do parser a partir da definição da gramática no arquivo .g4 (Lang.g4).

Compilar o Parser

```
make gen-parser
```

Compilar o Interpretador

```
make gen-interpreter
```

Compilar a Árvore de Sintaxe Abstrata

```
make gen-ast
```

Execução

Para executar o compilador e interpretar um arquivo específico, utilize:

```
java -cp $(ANTLR_JAR):. lang.LangCompiler -i <caminho_para_arquivo>
```

Onde `<caminho_para_arquivo>` deve ser substituído pelo caminho do arquivo Lang que deseja compilar e executar.

Testes

Para executar testes automatizados sobre o parser ou o interpretador:

```
make run-test-parser
make run-test-it
```